jamk.fi

Spring Security -kehyksen autentikointi- ja autorisointiratkaisujen implementointi

Markus Malm

Opinnäytetyö
Elokuu 2018
Tekniikan ja liikenteen ala
Insinööri (AMK), ohjelmistotekniikan koulutusohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Malm, Markus	Opinnäytetyö, AMK	Elokuu 2018
	Sivumäärä 40	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Spring Security -kehyksen autentikointi- ja autorisointiratkaisujen implementointi

Tutkinto-ohjelma
Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja(t)
Ari Rantala

Toimeksiantaja(t)
Combitech OY

Tiivistelmä

Opinnäytetyön tavoitteena oli tehdä selvitys Spring Securityn autentikointi- ja autorisointiratkaisujen implementoinnista olemassa olevaan ympäristöön. Selvityksellä haluttiin tuottaa Combitech OY:lle dokumentaatiota tukemaan sellaisia tilanteita, joissa Spring Securityn ominaisuuksia halutaan hyödyntää valmiissa ympäristössä.

Implementoinnin kohdeympäristöksi rakennettiin oluttietokanta-verkkosovellus. Ympäristön piti täyttää tiettyjä vaatimuksia, jotta sen perusteella tehdyn selvityksen tuloksia olisi mahdollista soveltaa tulevissa projekteissa. Verkkosovellus on toteutettu Javalla ja siinä hyödynnetään Spring Boottia ja sen sulautettua Tomcat-palvelinta sovelluksen ajamisessa. Verkkosovellus on rakennettu MVC-mallin mukaan ja sen view-tason sivu-templatet on toteutettu Thymeleafilla. Sovellusta tukeva tietokanta on toteutettu PostgreSQL:llä ja tietokannan dataa käsitellään sovelluksessa JPA:n ja Hibernate ORM:n avulla.

Työn lopputuloksena saatiin tuotettua verkkosovellus, jonka sisäisten service-rajapintojen ajo-oikeuksia on rajoitettu käyttäjien roolien perusteella hyödyntäen Spring Securityn metoditason autorisointiominaisuuksia. Lisäksi saatiin tuotettua dokumentaatiota, jota voidaan käyttää tulevaisuuden projekteissa tukena Spring Securityn ominaisuuksien implementoinnissa, jos niitä vaativia tilanteita tulee vastaan. Spring Securityn autentikointi- ja autorisointiratkaisujen implementointi oli helppoa ja suoraviivaista. Ratkaisut olivat helposti mukautettavissa käyttäjän tarpeisiin. Lisäksi Spring Security tarjoaa valmiita toteutuksia kolmansien osapuolien ratkaisuille, kuten OAuth2 tai LDAP.

Avainsanat (asiasanat)

Hibernate ORM, Java, JPA, Spring, Spring Security

Muut tiedot

jamk.fi

Description

Author(s)	Type of publication	Date
Malm, Markus	Bachelor's thesis	August 2018
		Language of publication: Finnish
	Number of pages: 40	Permission for web publication: x

Title of publication

Implementation of Spring Security framework's authentication and authorization solutions

Degree programme

Software Engineering

Supervisor(s)

Rantala, Ari

Assigned by

Combitech OY

Abstract

The goal of this thesis was to write a report on the implementation of Spring Security's authentication and authorization solutions to an already existing environment. The report serves as supportive documentation for Combitech OY when they encounter situations where employing Spring Security's features is applicable.

Beer database web application was created to serve as a target environment for the implementation process. The environment had to meet specific requirements for the results of the report to be applicable in future projects. The application was created with Java and it utilizes Spring Boot and its embedded Tomcat server. It was built according to the MVC-model and employs Thymeleaf on the view level to produce the page templates. The application utilizes a PostgreSQL database for data storage with Hibernate ORM handling persistence and object mapping within the java program.

The result of this thesis was a web application where the execution of service interface's methods is authorized according to the role of the user by using the method security features of Spring Security. In addition, documentation that can be used as support when implementing Spring Security's features, was produced. The implementation of the authentication and authorization features in Spring Security was easy and straight forward. The solutions were also highly customizable to suit the needs of the user. Additionally, Spring Security provides support for third-party solutions like OAuth2 and LDAP.

Keywords/tags (subjects)

Hibernate ORM, Java, JPA, Spring, Spring Security

Miscellaneous

Sisältö

Kuviot

Sanasto

Apache Maven

XML-pohjaista konfiguraatioita käyttävä koontityökalu.

Autentikointi

Autentikointi eli todennus tarkoittaa käyttäjän henkilöllisyyden varmistamista.

Autorisointi

Autorisointi eli valtuutus tarkoittaa käyttäjän oikeuksien määrittelyä.

Controller

Controller-luokat vastaavat MVC-arkkitehtuurissa sivujen logiikasta, eli datan siirtämisestä näkymään

CRUD

Create, read, update ja delete. Tietokantojen hallinnan neljä perusmetodia.

DAO

Data access object. Rajapinta tietokannassa sijaitsevan tiedon käsittelyyn.

DTO

Data transfer object. Java-luokkia jotka sisältävät ylensä vain julkisia muuttujia ilman logiikkaa. Tarkoitettu tiedon siirtämiseen rajapintojen välillä.

HTML

Hypertext Markup Language. Markup-kieli, jota käytetään verkkosivuilla. HTML:ssä verkkosivun rakenne ja sisältö määritellään html-elementtien avulla.

IoC

Inversion of Control. Ohjelmistosuunnittelun periaate, jossa kehittäjän itse kirjoittamat osat saavat toiminnallisuutensa sovelluskehykseltä.

Java

Olio-ohjelmointikieli.

JavaEE

Java Enterprise Edition. Spesifikaatio enterprise-tason Java-ohjelmistojen sovelluskehitykseen.

JPA

Java Persistence API. Rajapintaspesifikaatio, jossa on määritetty relaatiodatan käsittelyä Java-ympäristöissä.

MVC-arkkitehtuuri

Model-view-controller on erityisesti verkkosovelluksissa suosittu arkkitehtuurityyli, jossa tietomallit, käyttöliittymä, eli näkymä ja kontrollerien ohjainlogiikka on erotettu toisistaan.

ORM

Object-relational-mapping. Ohjelmointitekniikka, joissa tietokantataulut kartoitetaan olioihin.

SpEL

Spring Expression Language. Springin lausekekieli, jolla on mahdollista manipuloida olioita ja suorittaa kyselyitä ajon aikana.

Spring Framework

Java-sovelluskehys ja IoC-kontti. Osa Spring tuoteperhettä.

Spring Security -kehys

Spring tuoteperheen osa, joka sisältää tietoturvaominaisuuksia, kuten autentikointi- ja autorisointiratkaisuja.

Spring Bean

Springin alustamia, koostamia ja hallinnoimia olioita.

SQL

Standard Query Language. Tietokannoissa käytetty standardisoitu kyselykieli.

XML

Extensible Markup Language. Merkintäkieli sekä -standardi tiedon kuvaamiseen.

1 Johdanto

1.1 Tausta

Ohjelmistokehityksessä ja muilla ohjelmistoalan haaroilla työskentely on hyvin usein vahvasti kollaboratiivista työtä. Yritykset eri toimialoilta tekevät paljon yhteistyötä vaihtelevien olosuhteiden puitteissa suorasti tai epäsuorasti. Asiakas on voinut tilata ja työllistää eri tahoja saman tuotekokonaisuuden eri osa-alueisiin, jolloin suunnittelussa, kehityksessä ja toimituksessa on otettava huomioon useita sidosryhmiä. Pelkästään yhtä tahoa työllistävässä projektissa voidaan käyttää kolmannen osapuolen tarjoamaa tuotetta, joka edelleen voi pitää sisällään vielä jonkin kauemman tahon tuotteita tai sovelluskehyksiä.

Vastaan tulee tilanteita, joissa tuotteen jatkokehitykseen, kuten liitännäisten tuottamiseen, suuntautuvissa projekteissa pitää suunnitella ja ottaa huomioon teknologioiden yhteensopivuuksia konfliktien välttämiseksi ja toimivuuden saavuttamiseksi. Joskus tällaisia tilanteita voi tulla vastaan kesken projektin asiakkaan lisätilausten myötä tai projektin vaatimusmäärittelyn kehittyessä ja muuttuessa.

Edellä kuvattujen kaltaisissa tilanteissa yksi hyvä vaihtoehto on tutkia ja testata mahdollisia ratkaisuja, olivat ne sitten uusia teknologioita tai jo käytössä olevien syvällisempää hyödyntämistä, toteuttamlla demo johonkin muuhun kuin kehitettävään tuotteeseen. Demo voidaan toteuttaa kehitysrepositorion uuteen haaraan tai täysin itsenäiseen ja tuotekehityksestä irroitettuun ympäristöön.

1.2 Combitech OY

Combitech on osana Saab-konsernia toimiva tekniseen konsultointiin erikoistunut yritys. Pohjoismaissa Combitech työllistää yli 1900 työntekijää, joista Suomessa on noin 70 eri toimipisteissä Espoossa, Tampereella, Jyväskylässä ja Säkylässä. Suomessa Combitech on erikoistunut kyberturvallisuuteen ja puolustusalan teknisiin ratkaisuihin ja palveluihin. Combitechin asiakkaita ovat pääasiassa puolustusvoimat, puolustusteollisuus, yritykset ja julkinen sektori. (Tietoja meistä n.d.)

1.3 Toimeksianto

Combitechillä haluttiin tehdä selvitys ja demonstraatio Spring Securityn eri ominaisuuksien implementoinnista olemassa olevaan tuotteeseen nojaavassa kehityksessä tai kehitysympäristössä. Tällaisesta ennakoivasti toteutettavasta selvitys- ja tutkimustyötä on useita hyötyjä. Selvitystä tekevä työntekijä pystyy tutustumaan mahdollisesti ennalta tuntemattomaan teknologiaan ja näin lisäämään omaa osaamistaan ja sitä kautta myös yrityksen valmiuksia ja vahvuuksia. Lisäksi saadaan tuotettua dokumentaatiota, jota voidaan hyödyntää jo käynnissä olevissa tai tulevaisuuden projekteissa, jos projektin tarpeet ja kyseisen selvityksen kohteen ominaisuudet kohtaavat.

Tässä nimenomaisessa selvityksessä haluttiin tutustua erityisesti Spring Securityn metoditason valtuutusominaisuuksiin ja niiden kustomointiin. Selvitystä varten täytyi toteuttaa implementoinnin kohdeympäristöksi yksinkertainen web-sovellus muutamien vaatimusten mukaan. Vaatimukset liittyivät pääasiassa käytettäviin teknologioihin ja sovelluksen yleiseen rakenteeseen. Näiden vaatimusten nojalla sovellus noudattaa MVC-mallia ja hyödyntää JPA:ta ja Hibernate ORM:ää. Tämän opinnäytetyön toteutusta kuvaava luku on jaettu niin, että ensimmäinen puolisko kuvaa implementaation kohdeympäristöksi luodun web-sovelluksen toteutusta ja toinen puolisko on etenevä kuvaus niistä vaiheista, jotka johtavat metoditasolla tapahtuvan valtuuttamisen hyödyntämiseen.

2 Menetelmät ja käytetyt teknologiat

2.1 Ohjelmointiympäristö

Ohjelmointiympäristöksi eri vaihtoehtoina olivat Eclipse, IntelliJ IDEA ja NetBeans, joista valittiin NetBeanssin versio 8.2, koska se oli tutuin ja haluttiin kokeilla NetBeanssin Gradle-liitännäistä.

NetBeans on avoimen lähdekoodin Java-ohjelmointiympäristö, jonka Sun Microsystems julkaisi vuonna 1999. NetBeans oli alunperin tšekkoslovakialaisen yliopiston opiskelijaprojekti nimeltään Xelfi, jonka ensimmäinen julkaisu oli vuonna

1997. Sun Microsystems hankki nimensä NetBeanssiksi muuttaneen Xelfin itselleen 1999 ja muutti sen avoimen lähdekoodin projektiksi samana vuonna. (A Brief History of NetBeans n.d.)

Käännöstyökaluksi valittiin Gradle. Gradle oli todella helppokäyttöinen, vaikka NetBeans tarvitsikin liitännäisen sen käyttämistä varten natiivituen puuttumisen vuoksi. Liitännäisenkin kanssa Gradlea oli miellyttävä käyttää ja esimerkiksi riippuvuuksien määrittely oli paljon selkeämpää ja luettavampaa esimerkiksi Apachen Mavenin xml-muotoisiin pom-tiedostoihin verrattuna.

Noin puolessa välissä web-sovelluksen toteutusta jouduttiin kuitenkin siirtymään Gradlesta Apachen Maveniin, koska Hibernate ORM:n konfigurointi ei onnistunut NetBeanssin ja Gradle-liitännäisen bugin takia, joka esti Java Persistence API:n käyttämän persistence.xml-tiedoston automaattisen generoinnin. Tiedoston manuaalinen luominen ei myöskään onnistunut, sillä bugi esti Gradlea löytämästä ja lukemasta kyseistä tiedostoa, vaikka se oli sijoitettu oikeaan paikkaan.

2.2 Spring Framework

Spring Framework on avoimen lähdekoodin sovelluskehys Java-ympäristöön, joka luotiin vaihtoehdoksi JavaEE:lle. Spring ei suoraan noudata JavaEE-alustan koko spesifikaatioita, vaan integroi osia siitä kuten servletit, JPA:n ja yleiset annotaatiot.

Spring Frameworkin voi ajatella tuoteperheeksi, joka koostuu useista eri projekteista. Spring Framework pitää sisällään ydinominaisuuksia, Spring Cloud tarjoaa työkaluja pilvi- ja mikropalvelukeskeisiin järjestelmiin. Spring Data keskittyy erilaisiin tietokantaratkaisuihin tarjoten niille yhteneväisen ratkaisun datan käsittelyyn. Spring Security taas pitää sisällään erilaisia ratkaisuja autentikointiin ja autorisointiin, jotka ovat helposti mukautettavissa. Spring Frameworkille kokonaisuutena on vaikea löytää vaihtoehtoisia ratkaisuja, mutta sen mukautettavuuden ansiosta sen eri osia voi usein korvata saman osa-alueen eri sovelluskehyksillä.

2.3 Spring Boot

Spring Boot -projektin idea on tehdä Spring-pohjaisten sovellusten luomisesta ja ajamisesta nopeaa ja helppoa. Spring Boot sisältää sulautetun Tomcat-pavelimen, joka mahdollistaa sovelluksen ajamisen suoraan omasta paketistaan ilman konfiguraatioita.

Spring Bootia käytetiin, koska se teki web-sovelluksen kehittämisestä ja testaamisesta erittäin nopeaa, sillä sovelluksen käynnistäminen ja ajaminen oli mahdollista suoraan IDE:n kautta. Lisäksi pienellä konfiguraatiolla oli mahdollista saada käyttöliittymästä vastaavien html-template-tiedostojen muutokset näkyviin suoraan ilman Tomcat-palvelimen uudelleenkäynnistämistä, mikä nopeutti kehitystä ja testaamista entisestään.

2.4 Spring Security

Spring Security on sovelluskehys, joka on kehitetty tarjoamaan tietoturvapuolen autentikointi- ja autorisointiratkaisuja erityisesti JavaEE- ja Spring-pohjaisiin sovelluksiin. Spring Securityn tarjoamat ratkaisut ovat helposti laajennettavissa ja mukautettavissa käyttäjän tarpeisiin, mikä on yksi sen suurimmista vahvuuksista.

JavaEE:n spesifikaatiot sisältävät myös tietoturvaominaisuuksia, mutta usein ne eivät ole tarpeeksi laajoja kattamaan enterprise-tason sovelluksille tyypillisiä tarpeita. Näiden spesifikaatioiden mukaiset standardit eivät myöskään ole liikuteltavia pakettitasolla, minkä vuoksi palvelinympäristön vaihto johtaa yleensä kohdeympäristön työlääseen ja aikaa vievään uudelleenkonfiguraation. Spring Securityssä tällaiset ongelmat on onnistuttu ratkaisemaan. (Alex, Taylor, Winch, Hillert, Grandja & Bryant 2017.)

Spring Security sisältää myös paljon kolmansien osapuolien ratkaisujen implementaatioita sen omien ratkaisujen lisäksi. Spring Security voi käyttää tunnistautumisessa muun muassa LDAP:ia tai Kerberosta. Tästä syystä Spring Securityn käyttöönotto jo olemassa olevissa järjestelmissä on varteenotettava vaihtoehto, sillä aikaisempia toteutuksia voi olla mahdollista hyödyntää. Jo käytössä olevien teknologioiden integroimisella saadaan nopeutettua

implementointiprosessia ja voidaan välttää esimerkiksi ympäristömuutoksia, jotka voivat olla kalliita ja aikaa vieviä.

Mukautuvuutensa ansiosta Spring Security on hyvä ratkaisu myös silloin, jos valmiista ratkaisuista ei löydy sopivia vaihtoehtoja tai käytössä on legacy-järjestelmä, joka ei vastaa mitään olemassa olevia standardeja. Omien ratkaisujen toteuttaminen Spring Securityn päälle on varsin helppoa ja hyvä vaihtoehto tietoturvatarpeiden täyttämiseen.

Tässä toimeksiannossa Spring Securityyn haluttiin keskittyä juuri sen mukautuvuuden takia. Sevityksellä halutaan nimenomaan testata ja tutkia Spring Securitylla itse toteutettuja ratkaisuja olemassa oleviin ympäristöihin.

2.5 Thymeleaf

Thymeleaf on avoimen lähdekoodin Java-pohjainen XML ja HTML template-moottori, jolla on mahdollista esittää sisältöä niin paikallisesti, kuin verkkoympäristössäkin. Tässä opinnäytetyössä kuvatun Spring Security -implementaation kohdeympäristönä toimivassa MVC-mallilla toteutetussa verkkosovelluksessa Thymeleaf vastaa MVC-mallin ”View”-osuudesta. Lisäksi Thymeleaf tukee täysin Spring Frameworkia ja sisältää muutamia Spring Securityyn ja näkyvyyksien rajoittamiseen liittyviä ominaisuuksia, mitä käyn läpi tämän raportin Spring Securityn implementointia kuvaavassa osioissa.

Thymeleaf on kokonaisvaltainen vaihtoehto JSP:lle. JSP:stä poiketen Thymeleaf käyttää pelkästään tuttuja HTML-tageja JSP:n omien tagien sijaan. Muuttujia ja määrittelyitä on Thymleafin kanssa mahdollista kirjoittaa tagien sisään. Näitä käsitellään sitten controller-luokkien kautta kun sivu ladataan. Spring Frameworkin kanssa käytettynä määrittelyitä voidaan kirjoittaa SpEL:llä, mutta Thymeleafilla on myös oma kieli, joka vastaa pitkälti SpEL:llää. (Spring MVC view layer: Thymeleaf vs. JSP n.d.)

Vertailtaessa Thymeleafia ja muita template-moottoreita, kuten Apachen Velocityä erot näkyvät tuotteiden toiminnallisuudessa ja filosofiassa. Thymeleafin mainostetuimpia ominaisuuksia on ”natural templating”, jolla tarkoitetaan sitä, että sivut ovat itsenäisestikkin toimivia prototyyppejä. HTML-muotoiset template-sivut

voidaan näyttää selaimella sellaisenaan, niin että ne näyttävät oikeilta, ilman että niitä ajetaan Thymeleafin kautta. (About Thymeleaf n.d.) Tämä tekee kehittämisesti miellyttävää ja suoraviivaista, koska sivun ulkonäköä voi tarkastella avaamalla sen selaimella luottaen siihen, että näkymä vastaa sitä miltä sivu tulee näyttämään Thymeleafilla ajettuna.

Toiminnalliset erot liittyvät tuotteiden arkkitehtuurieroihin. Velocity on tekstiprosessori ja Thymeleaf on markup-parseri. Tämän ansioista Thymeleafilla on markup-kieli-pohjaisissa ympäristöissä ominaisuuksia, jotka eivät Velocityllä ole mahdollisia. (About Thymeleaf n.d.)

Valitsin Thymeleafin sen Spring Framework yhteensopivuuden kanssa. Template-sivujen kirjoittaminen vaati aluksi totuttelua, sillä SpEL:än sekoittaminen HTML-tagien sisään voi näyttää paikoin sekavalta, eikä se välttämättä oli erityisen luettavaa. Kun tähän tyyliin tottuu on dynaamisten sivujen luominen kuitenkin helppoa ja suoraviivaista.

2.6 PostgreSQL

PostgreSQL on avoimen lähdekoodin olio-relaatiotietokantajärjestelmä. Postgres noudattaa pääosin SQL-standardia, mutta poikkeaa siitä silloin, kun se on ristiriidassa Postgressin omien ominaisuuksien kanssa. Postgres mahdollistaa omien tietotyyppien ja mukautettujen funktioiden luomisen.

Valitsin PostgreSQL:n, koska minulla ei ollut siitä aikaisempaa kokemusta. Tässä opinnäytetyössä Postgres-tietokanta pyörii lokaalisti toteutuksen testaamisen helpottamiseksi. Tietokannan hallintaan käytin pgAdmin-hallintatyökalua.

2.7 Hibernate ORM

Hibernate on object-relational mapping -kehys java-ympäristöihin. Hibernate ja ORM-työkalut yleensä mahdollistavat java-luokkien kartoittamisen tietokanta-tauluihin ja Javan tietotyyppien kartoittamisen SQL-tietotyyppeihin. Tässä opinnäytetyössä yksi kohdeympäristön vaatimuksista oli ORM:n hyödyntäminen ja päädyin Hibernateen koska se oli entuudestaan tuttu.

Hibernate on versiosta 3.2 eteenpäin sisältänyt JPA-standardin eli Java Persistence API:n implementaation, joten sen on mahdollista käyttää sen omien funktioiden lisäksi Javan oman JPA-rajapinnan funktioita. Näin on mahdollista muun muassa hyödyntää JPA:n Criteria API:a tietokantakyselyissä sekä JPA-standardin mukaisia annotaatioita entity-luokissa ja persistence-rajapinnoissa.

Hibernatella on mahdollista automatisoida paljon tietokannan taulujen relaatioihin liittyviä asioita, jolla voidaan vähentää koodia. Hunona puolena voi mainita sen, että ympäristö, jossa ORM-tyylistä tietokantaratkaisua käytetään, on erittäin riippuvainen työkalusta, jolla ORM toteutetaan. Lisäksi ohjelmoija on vastuussa siitä, että data pysyy tietokannan ja asiakaspään välillä yhteneväisenä ja synkronoituna. Tämä on usein todella haasteellista varsinkin monimutkaisemmissa tietokannoissa.

3 Oluttietokanta-verkkosovellus

3.1 Sovelluksen kuvaus

Tässä opinnäytetyössä kuvatun Spring Security -selvityksen kohdeympäristöksi tuotettiin pieni oluttietokanta-sovellus, johon käyttäjät voivat lisätä juomiaan oluita ja niiden arvosteluja. Käyttäjät kuuluisivat eri ryhmiin ja voisivat nähdä vain omia tai muiden samaan ryhmään kuuluvien käyttäjien olutarvioita. Ajatuksena oli, että käyttäjien pääsyä ja valtuuksia eri toimintoihin rajoitettaisiin käyttäjille määriteltyjen roolien perusteella. Näiden toimintojen logiikasta vastaavien funktioiden ajamista pyrittäisiin lopulta rajoittamaan Spring Securityn metoditason valtuuttamisominaisuuksilla.

Verkkosovellus on rakennettu Spring Bootilla, Spring MVC:llä ja Thymeleafilla, missä Spring on vastuussa sovelluksen logiikasta ja controller-luokista Thymeleafin tuottaessa HTML-muotoiset sivunäkymät. Sovelluksen rakentaminen Spring Bootin päälle nopeutti ja suoraviivaisti kehitystä, minkä ansioista sovellus saatiin rakennettua nopeasti ja vaivattomasti. Näin päästiin hyvin pienellä vaivalla suoraan itse Spring Securityn ominaisuuksien implementoinnin pariin.

Verkkosovellukselle määriteltiin jotain vaatimuksia sen rakenteen ja käytettyjen teknologioiden suhteen. Tärkein näistä oli, että tietokantatoeteutus noudattaa ORM-

mallia. Back- ja frontendin pitää myös olla riippumattomia toisistaan, koska todellisuudessa niitä usein ajetaan toisistaan erillisissä ympäristöissä. Tässä sovelluksessa tämä ei toteudu, sillä kaikki toiminnallisuus pyörii paikallisesti samalla koneella. Asia on kuitenkin otettu huomioon kehityksessä siten, että tietokannasta tuotuja entity-luokkien olioita ei käsitellä sellaisinaan muuta kuin backendin dao-luokissa. Entityt siirretään aina DTO-luokissa frontendille, jossa ne muutetaan entity-luokkia vastaavien UI-luokkien olioiksi datan käsittelyä ja näyttämistä varten. Näin front- ja backend ovat yhteydessä toisiinsa vain DTO-luokkien olioita liikuttelevien service-rajapintojen kautta.

3.2 Projektin luonti

Projektin luominen NetBeanssilla tapahtui kuten tavallisestikin. Lisäksi määriteltiin Spring Bootin riippuvuudet mavenin pom.xml-tiedostoon, kuten kuviossa 1.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.10.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
```

Kuvio 1. pom.xml

Tässä vaiheessa projektin kääntämiseen käytettiin vielä Gradlea, koska Apachen Maveniin vaihtamiseen johtanut bugi ei vielä ollut tullut vastaan. Mavenin pom.xml-tiedostoa vastaava Gradlen build.gradle-tiedosto tekee käytännössä samat asiat, mutta sen syntaksi on erilaista.

Spring Boottia varten luotiin kuvion 2 mukainen Application.java-niminen pääluokka, joka toimii ohjelman entry pointtina.

```
package fi.combitech.beermanager;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 *
 * @author udmama
 */
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Kuvio 2. Application.java

Spring Boot osaa tehdä kaikki tarvittavat konfiguraatiot automaattisesti @SpringBootApplication-annotaation avulla. Annotaatio pitää periaattessa sisällään kolme muuta annotaatiota niiden oletusasetuksilla: @EnableAutoConfiguration, @ComponentScan ja @Configuration. @EnableAutoConfiguration antaa Springin tehdä itse tarvittavat konfiguraatiot, @ComponentScan mahdollistaa @Component-annotoitujen luokkien löytämisen paketin sisältä ja @Confugration mahdollistaa uusien beanien rekisteröimisen ja konfiguraatioluokkien tuomisen.

Seuraavaksi luotiin tähän astisen konfiguraation toimivuuden testaamista varten kuvion 3 mukainen controller-luokka nimeltään GreetingController, joka vastaa datan näyttämisestä käyttäjälle. Käyttäjä voi asetta parametrina nimensä selaimen osoiteriville, mutta nimen puuttuessa arvo on oletuksena ”World!”.

```
@Controller
public class GreetingController {
    @RequestMapping("/greeting")
    public String greeting(@RequestParam(value="name",
            required=false, defaultValue="World!") String name, Model model) {
        model.addAttribute("name", name);
        return "greeting";
    }
}
```

Kuvio 3. GreetingController

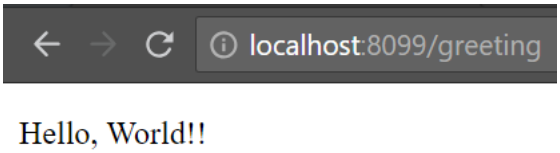
@Controller-annotaatio kertoo Springille, että kyseinen luokka on controller-luokka. @RequestMapping-annotaatio taas kertoo Springille, että kun käyttäjä navigoi verkkosivuston greeting-polkuun, sen tulee ajaa greeting()-funktio joka, täyttää greeting.html-templatesivun. Template-sivu sisältää vain sanan ”World” ja paikan, johon controller-luokka asettaa parametrina saamansa arvon.

Seuraavaksi luotiin projektin src/main/resources-polkuun application.properties-tiedoston, mihin määritellään sovelluksen asetuksia. Tiedostoon lisättiin kuvion 4 mukainen rivi ”server.port=8099”, joka kertoo Springille, missä portissa sovellus pyörii.

```
server.port=8099
```

Kuvio 4. application.properties

Kun nyt sovellus käynnistetään NetBeanssin sisältä, se lähtee pyörimään Spring Bootin sulautetussa Tomcat-palvelimessa. Navigoidessa selaimella osoitteeseen ”localhost:8099/greeting” nähdään tyhjä sivu, jolla lukee ”Hello World!”, kuten kuviossa 5. Nyt kun Spring Boot -ympäristön toimivuus on todettu, voidaan lähteä rakentamaan verkkosovellusta.



Kuvio 5. Hello World!

3.3 Sivu-templatet

Seuraavaksi tehtiin ensimmäiset sivu-templatet Thymeleafilla. Springiltä löytyy oma artefakti Thymeleafille, joka lisätään pom.xml:ään. Luodaan ”src/main/resources”-polkuun uusi ”templates”-kansio, jonne Thymeleafin HTML-template-tiedostot sijoitetaan. Templates-kansioon luotiin uusi HTML-tiedoston index.html kuten kuviossa 6, joka toimii sovelluksen aloitussivuna. Lisäksi luotiin header- ja footer-tiedostot, joita voidaan käyttää index-sivulla sekä kaikilla tulevilla sivuilla.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Beer manager</title>
    </head>
    <body>
        <header th:replace="header :: header"></header>
        <h1>Welcome</h1>
        <p> Time: <b th:text="${execInfo.now.time}"></b></p>
    </body>
    <div th:replace="footer :: footer"></div>
</html>
```

Kuvio 6. index.html

Aloitussivu näyttää yksinkertaisen tervetulotoivotuksen sekä tällä hetkellä vielä tyhjän headerin ja footerin, jotka on tuotu index-sivulle Thymeleafin ”th:replace”-lauseella. Sivu näyttää myös tämänhetkisen kellonajan Thymeleafin ”exe-

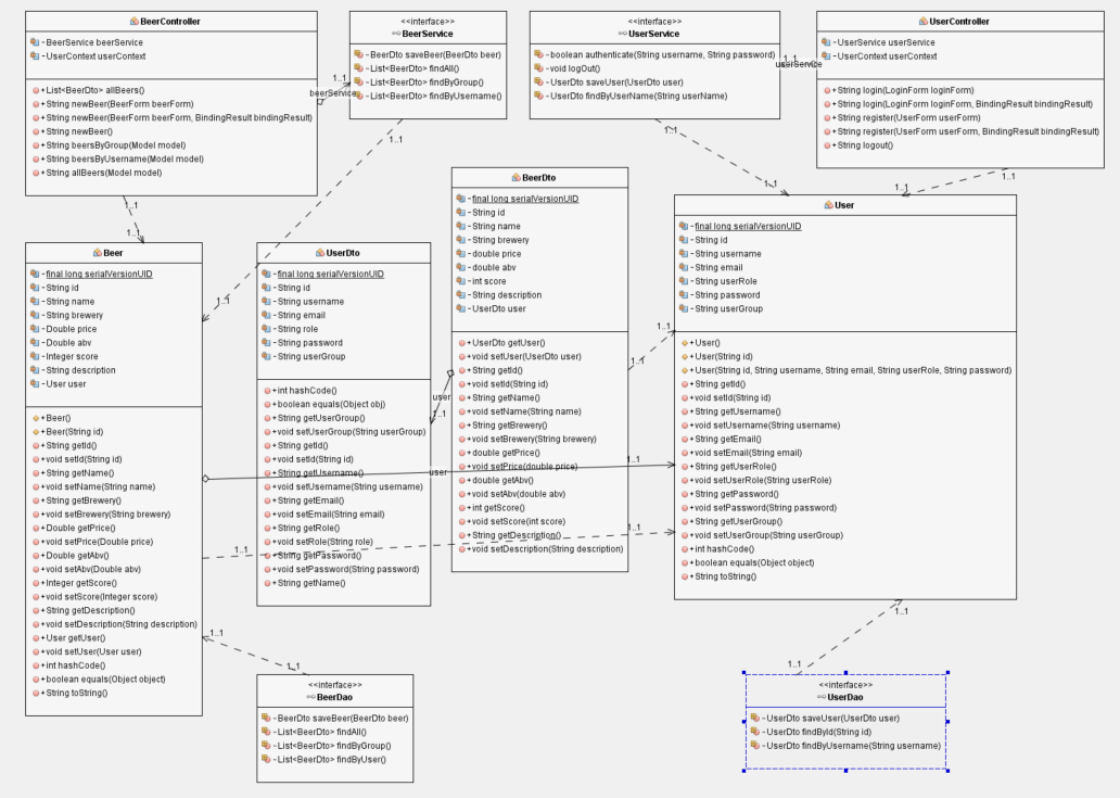
cInfo.now.time”-lauseella. Sivusto tulee käyttämään header-templatea navigointi-palkkina ja siihen lisätään linkkejä uusille sivuille sitä mukaa, kun niitä tehdään. Jotta localhost:8099 ohjaisi sovelluksen aloitussivulle, tarvitsee vain tehdä pieni muutos WelcomeControlleriin, jotta se on kuvion 7 mukainen.

```
@Controller
public class WelcomeController {
    @RequestMapping("/")
    public String index() {
        return "index";
    }
}
```

Kuvio 7. WelcomeController.java

3.4 Beer- ja User-luokat

Sovelluksessa käytettiin kahdenlaisia olioita. Beer-luokan oliot ovat tietokantaan tallennettavia olioita, jotka sisältävät myös oluen tallentaneen käyttäjän sekä arvion oluesta. User-luokan olilot sisältävät tiedon käyttäjästä, käyttäjän roolista ja ryhmästä johon käyttäjä kuuluu. Kuviossa 8 on kuvattu verkkosovelluksen tärkeimmät luokat, Controller-luokat ja rajapinnat UML-diagrammissa.



Kuvio 8. UML-diagrammi

Lisäksi molempia luokkia varten luotiin niitä vastaavat DTO-luokat, joita käytetään tiedon kuljettamiseen käyttöliittymän ja tietokannan välillä. DTO:t ovat dummy-luokkia, jotka sisältävät vain julkisia muuttuja. Niiden ainoa tehtävä on toimia tiedon kuljettajina.

Tässä vaiheessa tehtiin myös template-sivut oluiden tallentamista ja käyttäjien luontia varten. Luontilomakkeita varten tarvitaan BeerForm- ja UserForm-luokat joiden avulla template-sivujen lomakkeisiin syötetyt tiedot saadaan siirrettyä controller-luokille. Samalla luotiin BeerController- ja UserController-luokat, jotka vastasivat kaikkien nimiensä mukaisten olioiden käsittelystä asiakaspäässä.

Uuden oluen luomiseen tarkoitetutussa näkymässä voidaan tarkastella, kuinka Thymeleafilla on mahdollista käsitellä lomakkeita. Kuviossa 9 on esitetty, miten HTML:n form-tagin sisällä määritellään Controller-luokassa ajettava funktio sekä käsiteltävä olio, tässä tapauksessa beerForm. Input-tagien sisällä th:field-lauseella sidotaan kenttien arvot beerForm-olion muuttujiin.

```
<!DOCTYPE html>

<html>
    <head>
        <title>New beer</title>
    </head>
    <body>
        <header th:replace="header :: header" />
        <h1>Add new beer</h1>
        <form action="#" th:action="@{/newbeer}" th:object="${beerForm}" method="post">
            <div><label for="name">Name</label></div>
            <input type="text" th:field="*{name}" />
            <div><label for="brewery">Brewery</label></div>
            <input type="text" th:field="*{brewery}" />
            <div><label for="price">Price</label></div>
            <input type="number" step="0.01" th:field="*{price}" />
            <div><label for="ab">Alcohol content</label></div>
            <input type="number" step="0.1" th:field="*{abv}" />
            <div><label for="score">Score</label></div>
            <input type="number" th:field="*{score}" />
            <div><label for="description">Description</label></div>
            <textarea th:field="*{description}" />
            <input type="submit" value="Add beer" />
        </form>
    </body>
</html>
```

Kuvio 9. newbeer.html

Kuviossa 10 BeerController-luokan lomakkeen lähettämistä vastaavassa funktiossa luodaan uusi UIBeer-olio beerForm-lomakkeesta saatujen arvojen perusteella. Tässä vaiheessa ei vielä kuitenkaan uudella UI-oliolla tehdä mitään, sillä tiedon kuljettamisesta ja tallentamisesta vastaavia service-luokkia ei ole vielä implementoitu.

```
    @RequestMapping("/newbeer")
    public String newBeer(BeerForm beerForm) {
        return "newbeer";
    }

    @RequestMapping(value = "/newbeer", method = RequestMethod.POST)
    public String newBeer(@Valid BeerForm beerForm, BindingResult bindingResult) {
        BeerDto newBeer = new BeerDto();

        newBeer.setName(beerForm.getName());
        newBeer.setBrewery(beerForm.getBrewery());
        newBeer.setPrice(beerForm.getPrice());
        newBeer.setAbv(beerForm.getAbv());
        newBeer.setScore(beerForm.getScore());
        newBeer.setDescription(beerForm.getDescription());
        newBeer.setUser(userContext.getCurrentUser());

        beerService.saveBeer(newBeer);
        return "newbeer";
    }
```

Kuvio 10. beerController & /newbeer-mappaus

Käyttäjiä varten luodut luokat ovat samanlaisia, mutta lomakkeet vastaavat UIUser-luokan olioita ja tietoja käsitellään UserController-luokassa. Toimintaperiaate on user- ja beer-luokkien olioiden kanssa sama.

3.5 Tietokanta

Käyttäjä- ja olut-luokkien olioita tulee pystyä tallentamaan pysyvästi. Tätä varten perustetaan PostgreSQL-tietokanta. Luodaan PostgreSQL-tietokannan hallinnoimiseen tarkoitetulla pgAdmin-työkalulla uusi database, jonka sisälle luodaan oma schema oluttietokantaa varten. Tehdään tietokantaan kuvioiden 11 ja 12 mukaiset taulut oluille ja käyttäjille.

```
-- Table: beermanager.beer

-- DROP TABLE beermanager.beer;

CREATE TABLE beermanager.beer
(
    id character varying(50) COLLATE pg_catalog."default" NOT NULL,
    name character varying(50) COLLATE pg_catalog."default",
    brewery character varying(50) COLLATE pg_catalog."default",
    price double precision,
    abv double precision,
    score integer,
    description character varying(250) COLLATE pg_catalog."default",
    CONSTRAINT beer_pkey PRIMARY KEY (id)
)
WITH (
    OIDS = FALSE
)
TABLESPACE beermanager;

ALTER TABLE beermanager.beer
    OWNER to postgres;
```

Kuvio 11. beer-taulun SQL

```
-- Table: beermanager."user"

-- DROP TABLE beermanager."user";

CREATE TABLE beermanager."user"
(
    id character varying(50) COLLATE pg_catalog."default" NOT NULL,
    username character varying(20) COLLATE pg_catalog."default" NOT NULL,
    email character varying(50) COLLATE pg_catalog."default" NOT NULL,
    user_role character varying COLLATE pg_catalog."default" NOT NULL,
    password character varying(20) COLLATE pg_catalog."default" NOT NULL,
    user_group character varying COLLATE pg_catalog."default",
    CONSTRAINT user_pkey PRIMARY KEY (id)
)
WITH (
    OIDS = FALSE
)
TABLESPACE beermanager;

ALTER TABLE beermanager."user"
    OWNER to postgres;
```

Kuvio 12. user-taulun SQL

Lisäksi tehdään valmiiksi kuvion 13 mukaisen user_beer-taulun Hibernaten relaatiokartoitusta varten.

```
-- Table: beermanager.beer_user

-- DROP TABLE beermanager.beer_user;

CREATE TABLE beermanager.beer_user
(
    beer_id character varying COLLATE pg_catalog."default" NOT NULL,
    user_id character varying COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT beer_user_pkey PRIMARY KEY (beer_id, user_id)
)
WITH (
    OIDS = FALSE
)
TABLESPACE beermanager;

ALTER TABLE beermanager.beer_user
    OWNER to postgres;
```

Kuvio 13. beer-user -taulun SQL

Lisätään kuvion 14 mukaisesti tietokanta datasourceksi application.properties-tiedostoon, jotta sovellus on tietoinen sen olemassa olosta ja voi olla yhteydessä siihen.

```
spring.datasource.maxActive=5
spring.datasource.url = jdbc:postgresql://localhost:5432/testDB
spring.datasource.username = postgres
spring.datasource.password = adpg
```

Kuvio 14. application.propertiesin datasource-määrittelyt

3.6 Hibernaten käyttöönotto

Kun tietokanta on perustettu, voidaan konfiguroida ja ottaa Hibernate käyttöön. Lisätään tarvittavat hibernate-core- ja hibernate-entitymanager-riippuvuudet pom.xml-tiedostoon. NetBeanssilla voi automaattisesti generoida entity-luokat, joihin Hibernate kartoittaa tietokannan taulut. Beer.javan entity-luokka on näkyvissä liitteessä 1. User.java-luokka generoidaan samalla tavalla.

Entity-luokkien generoinnin yhteydessä NetBean luo automaattisesti kuvio 15 mukaisen persistence.xml-tiedoston, jossa on määritelty entity-luokat sekä tietokannan tiedot, jotka Hibernate tarvitsee toimiakseen.

```
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="fi.combitech_beermanager_jar_1.0-SNAPSHOTPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>fi.combitech.beermanager.hibernate.db.Beer</class>
    <class>fi.combitech.beermanager.hibernate.db.User</class>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/testDB"/>
      <property name="javax.persistence.jdbc.user" value="postgres"/>
      <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver"/>
      <property name="javax.persistence.jdbc.password" value="adpg"/>
    </properties>
  </persistence-unit>
</persistence>
```

Kuvio 15. persistence.xml

Yksinkertaisimmillaan Hibernaten käyttöönotto olisi tässä, mutta koska Hibernaten halutaan automaattisesti ylläpitävän Beer- ja User-olioiden välisiä relaatioita joudutaan siitä vastaavat osuudet kirjoittaman käsin.

Lisätään Beer-entity-luokalle OneToOne-relaation annotaatio kuvion 16 mukaisesti. @JoinTable annotaation sisällä kerrotaan mitä taulua ja mitä sarakkeita Hibernate käyttää relaatioiden kirjaamiseen ja ylläpitämiseen. Nämä annotaatiot on asetttu User-entity-luokan muuttujalle. Koska tämä suhde on yksipuolinen, eikä käyttäjien tarvitse toiminnallisuuden vuoksi tietää mitkä Beer-entityt liittyvät mihinkin käyttäjään User-entity-luokkaan ei tarvitse lisätä mitään.

```
@OneToOne
@JoinTable(name="beer_user", schema = "beermanager",
joinColumns = @JoinColumn(name="beer_id", referencedColumnName="id",
        unique=true, nullable=true, insertable=true, updatable=true),
        inverseJoinColumns = @JoinColumn(name="user_id", referencedColumnName="id"))
private User user;
```

Kuvio 16. Beer @OneToOne

3.7 DAO-luokat ja service-rajapinnat

Jotta tietoa voidaan tallentaa tietoa ja hakea tietokannasta Hibernaten avulla tarvitaan DAO-luokat kaikkia entity-luokkia varten. DAO-luokan avulla muutetaan entity-luokan oliot DTO-luokan olioiksi asiakaspäähän siirtymistä varten. DAO-luokat ovat vastuussa myös vastakkaiseen suuntaan menevistä operaatioista, eli DTO-luokkien olioiden muuttamisesta entity-luokan olioiksi kantaan tallentamista varten.

Lisäksi DAO-luokat sisältävät metodit CRUD-operaatioita varten. JPA:lla tietokantakutsut toteutetaan Entity Managerin kautta. Yksinkertaisimmassa hakukutsuissa voidaan käyttää automaattisesti generoituja named queryita, jotka löytvät entity-luokista. Kuviossa 17 näkyy, kuinka DAO-luokassa käytetään automaattisesti luotua findAll NamedQueryä.

```
@Override
public List<BeerDto> findAll() {
    List<BeerDto> results = new ArrayList<>();

    try {
        TypedQuery<Beer> q = entityManager.createNamedQuery("Beer.findAll", Beer.class);
        List<Beer> beerList = q.getResultList();
        beerList.forEach((beer) -> {
            results.add(dbToDomain(beer));
        });
    } catch (Exception e) {

        e.getStackTrace();
    }
    return results;
}
```

Kuvio 17. BeerDaoImpl-luokan findAll()-metodi

Syvällisemmissä hakukutsuissa voidaan entity managerin ja criteria builderin avulla luoda omia kyselylauseita ja määritellä sitten lauseelle parametreja ja join-lausekkeita. Kuviossa 18 on esitelty metodi, joka hakee oluet kirjautuneen käyttäjän userGroup-muuttujan perusteella.

```
@Override
public List<BeerDto> findByGroup() {

    List<BeerDto> results = new ArrayList<>();
    CriteriaBuilder cb = entityManager.getCriteriaBuilder();
    CriteriaQuery<Beer> q = cb.createQuery(Beer.class);
    Root<Beer> r = q.from(Beer.class);



    Join<Beer, User> joinRoot = r.join("user");
    Predicate andPredicate = cb.like(cb.upper(joinRoot.get("userGroup").as(String.class)),
            (userContext.getCurrentUser().getUserGroup().toUpperCase()));
    q.select(r);
    q.where(cb.and(andPredicate));

    TypedQuery<Beer> query = entityManager.createQuery(q);
    List<Beer> res = query.getResultList();

    res.forEach((beer) -> {
        results.add(dbToDomain(beer));
    });

    return results;
}
```

Kuvio 18. BeerDaoImpl-luokan findByGroup()-metodi

Nyt tarvitaan vain service-rajapinta DAO-luokkien ja asiakaspään väliin, jotta voidaan suorittaa hakuja ja tallennusoperaatioita sovelluksen verkkokäyttöliittymästä käsin. Oluille ja käyttäjille tehdään omat service-luokat, joiden kautta DAO-luokkien metodeita kutsutaan. Tarvittaessa voidaan lisätä service-luokkiin geneerisä metodeja esimerkiksi käyttäjän uloskirjaamista varten.

4 Spring Security

4.1 Autentikointi ja käyttäjät

Kohdeympäristö on nyt siinä kunnossa, että Spring Securityn ominaisuuksien implementointi voidaan aloittaa. Ensiksi kuitenkin tarvitaan tapa jolla pidetään kirjaa tällä hetkellä sisään kirjautuneesta käyttäjästä. Tätä varten luodaan kuvion 19 mukainen UserContext-rajapinta, jolla voidaan asettaa ja hakea sisäänkirjautunut käyttäjä.

```
package fi.combitech.beermanager.services;

import fi.combitech.beermanager.dto.UserDto;

/**
 *
 * @author udmama
 */
public interface UserContext {

    UserDto getCurrentUser();

    String getCurrentUserUsername();

    void setCurrentUser(final UserDto user);

}
```

Kuvio 19. UserContext.java

Spring Security tarjoaa monia eri vaihtoehtoja käyttäjien autentikointiin, mutta kaikki johtavat samaan tulokseen. Käyttäjiä voidaan verrata sisäiseen muistivarastoon tai ulkoiseen tietokantaan Springin JDBC-tuen ansioista. Spring Data -projekti

mahdollistaa JPA:n hydöntämisen ORM-tietokantatoteutusten kanssa. Lisäksi Spring Secrity on mahdollista sovittaa toimimaan esimerkiksi LDAP:n tai OAuth2:n rinnalla. Spring Security täyttää SecurityContextin Authentication-oliolla, joka edustaa kaikkea autentikointihetkellä käyttäjästä kerättyä tietoa. SecurityContext asetetaan sitten SecurityContextHolder-rajapinnalle, jonka on mahdollista päästä käsiksi kirjautuneeseen käyttäjään authentication-olion kautta. Tämän toteutuksen getCurrentUser()-metodi on kuvattu kuviossa 20.

```
public User getCurrentUser() {
    SecurityContext context = SecurityContextHolder.getContext();
    Authentication authentication = context.getAuthentication();
    if (authentication == null) {
        return null;
    }
    String userName = authentication.getName();
    return userDao.findByUsername(userName);
}
```

Kuvio 20. getCurrentUser()-metodi

Koska Spring ei ymmärrä sovelluksen oman User-luokan olioita, täytyy ne muuttaa sellaiseen muotoon, jota Spring Security ymmärtää ja pystyy käsittelemään. Sovelluksen User-olioiden perusteella luodaan Spring Securityn core.userdetails.user-luokan olioita, jotka sisältävät käyttäjän nimen, salasanan ja listan Spring Securityn GrantedAuthority-olioista, jotka ovat käyttäjälle asetettuja rooleja tai oikeuksia. Näitä User-olioita asetetaan Springin UserDetails-luokan olioille. GrantendAuthority olioiden perusteella voidaan antaa käyttäjälle valtuuksia ja tarkistaa niitä Springin hasRole- ja hasAuthority-funktioilla. GrantedAuthorityt ovat String-muotoisia ja roolit erotetaan valtuuksista ”ROLE”-etuliitteellä. Muuten merkkijonjen sisältö on kehittäjän päätettävissä.

Spring Security sisältää UserDetailsInterfacen, jota käytetään Spring User-luokan olioiden hakemiseen. UserDetailsManager on UserDetailsInterfacea laajentava luokka, jolla taas hoidetaan uusien User-luokan olioiden tallentaminen ja olemassa olevien päivittäminen.

Spring-käyttäjää kirjautuneeksi asetettaessa haetaan käyttäjän nimen perusteella UserDetailsService-rajapinnan kautta oikea käyttäjä ja luodaan sen mukaan Authentication-luokan olioin, joka asetetaan SecurityContextille.

4.2 Metoditason valtuutus

Spring Security tarjoaa ratkaisun, joka mahdollistaa sovelluksien metodien turvaamisen. Tällaiseen metodien turvaamiseen Spring Securitystä löytyy kaksi

annotaatio-pohjaista lähestymistapaa; preAuthorize- ja postAuthorize, joille molemmille on omat mahdolliset käyttökohteensa. PreAuthorize varmistaa, että asetetut ehdot täyttyvät ennen metodin ajoa. Kun ehdot eivät täyty, metodikutsu epäonnistuu. PostAuthorize taas tarkistaa, että ehdot täyttyvät edelleen metodin palautuessa. (Knutson, Winch & Mularien 2017, 317.)

Näiden avulla halutaan rajoittaa sovelluksen Service-luokkien metodien ajo-oikeuksia. Ensiksi testataan ajo-oikeuden rajoitusta suoraan kirjautuneena olevan käyttäjän GrantedAuthority-olioiden perusteella ja sen jälkeen kokeillaan rajoitussääntöjen mukauttamista kirjoittamalla oma SecurityService-rajapinta.

Jotta Pre- ja PostAuthorize-annotaatioita voidaan käyttää, täytyy ne sallia muokkaamlla AppConfig-konfiguraatioluokkaa, kuten kuviossa 21.

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class AppConfig {
```

Kuvio 21. @EnableGlobalMethodSecurity

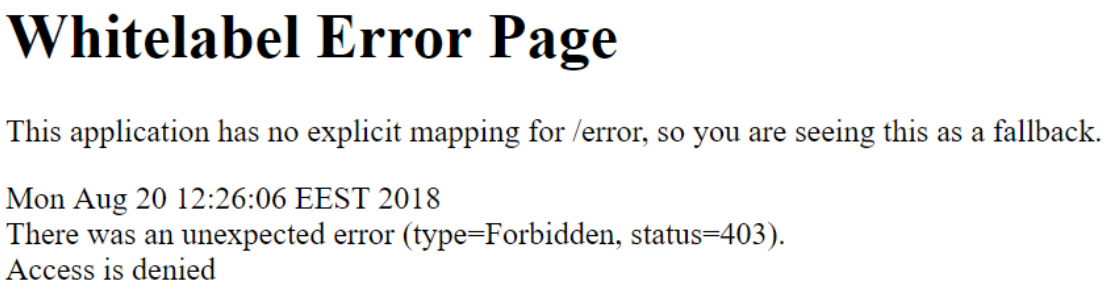
Lisäksi annotaatio täytyy asettaa suojattavalle metodille kuvion 22 mukaisesti.

```
    @Override
    @PreAuthorize("hasRole('ADMIN')")
    public List<BeerDto> findAll() {
        return beerDao.findAll();
    }
```

Kuvio 22. @PreAuthorize

Nyt sovellus tarkistaa onko kirjautuneen Springin UserDetails-luokan olion GrantedAuthority-listassa vastaavaa admin-roolia ja joko sallii tai estää metodin ajon tämän perusteella. Jos arvo löytyy haettaessa kaikkia oluita, niin oluet haetaan ja

esiteään käyttäjälle normaalisti. Jos arvoa ei löydy, käyttäjä näkee kuvion 23 mukaisen näkymän.



Kuvio 23. Access is denied

Acces Denied -sivua on mahdollista muokata ja ohjaus voidaan tehdä haluatessa omalle template-sivulle määrittelemällä se Controller-luokassa.

Oman valtuutustoteutuksen voi tehdä erittäin vaivattomasti. Aluksi kirjoitetaan oma kuvion 24 mukainen CustomSecurityService-rajapinta ja sen implementaatio, johon voidaan määritellä valtuutuksista vastuussa olevia funktioita.

```
package fi.combitech.beermanager.services;

/**
 *
 * @author udmama
 */
public interface CustomSecurityService {
    public boolean isAdmin();
}
```

Kuvio 24. CustomSecurityService.java

Oman SecurityService-toteutuksen testaamiseksi kirjoitettiin oma isAdmin()-funktion, kuten kuviossa 25. Funktio tekee käytännössä saman operaation, kuin

SpringSecurityn oman HasRole()-funktio, mutta tarkistus tehdään järjestelmän omalle käyttäjälle Springin UserDetails-luokkaan tallennetun käyttäjän sijaan.

```
    @Override
    public boolean isAdmin() {
        UserDto currentUser = userContext.getCurrentUser();
        return currentUser.getRole().equals("admin");
    }

}
```

Kuvio 25. CustomSecurityServiceImpl-luokan isAdmin()-metodi

Seuraavaksi rekisteröitiin CustomSecurityService kuvion 26 mukaisesti beaniksi AppConfig-luokassa. Nyt sitä ja sen metodeja voidaan kutsua SpEL-lauseella suoraan annotaation sisässä.

```
    @Bean
    public CustomSecurityService customSecurityService() {
        return new CustomSecurityServiceImpl();
    }
```

Kuvio 26. CustomSecurityServicen bean-rekisteröinti

Nyt tarvitsee vain käydä vaihtamassa vanha HasRole()-kutsu annotaation sisästä oman SecurityServicen isAdmin()-metodin kutsuun, kuten kuviossa 27. Admin-käyttäjänä kirjautuneena nähdään oletetusti kaikki oluet, mutta muilla käyttäjillä nähdään taas Acces Denied-sivun. Beanin kutsuminen suoraan teke omien valtuutusratkaisujen luomisesta todella helppoa ja monipuolista. Lisäksi tämä mahdolistaa omien ratkaisujen toteuttamisen olemassa oleviin ympäristöihin sillä oletuksella, että Pre- ja PostAuthorize-annotaatiot on sallittu SpringSecurityn SecurityConfig-konfiguraatioluokassa.

```
@Override
@PreAuthorize("@customSecurityService.isAdmin()")
public List<BeerDto> findAll() {
    return beerDao.findAll();
}
```

Kuvio 27. isAdmin()-metodin kutsuminen

Koska oluttietokannan template-sivut on toteutettu Thymeleafilla, voitaisiin käyttäjän näkymiä rajoittaa myös MVC-mallin view-tasolla. Thymeleafin integroidun Spring Security tuen ansiosta template-tiedostoissa voitaisiin tagien sisään määritellä valtuutussääntöjä, joilla olisi mahdollista piilotta elementtejä näkyvistä esimerkiksi silloin, kun käyttäjä ei ole autentikoitunut. Näitä sääntöjä voidaan määrittää suoraan templateihin tai hakea valtuutuksesta vastuussa oleva logiikka Controller-luokasta. (Knutson ym. 2017, 309-312.)

5 Tulokset

Opinnäytetyön tuloksena saatiin aikaan verkkosovellus, jolla demonstroitiin Spring Securityn metoditason valtuutusominaisuuksia. Liitteissä 2, 3 ja 4 on nähtävissä oluiden käsittelystä vastaava controller- ja DAO-luokka. Sovelluksen ruutukauppauksia on näkyvissä liitteissä 5, 6 ja 7. Sovelluksen ja Spring Security implementaation tuloksena syntyneessä selvityksessä pystyttiin todentamaan mukautettujen valtuutustoteutuksien toimivuus Spring Securityn kautta. Lisäksi pystyimme toteamaan, että niiden käyttäminen olemassaolevissa ympäristöissä on mahdollista sillä oletuksella, että Spring Security on ympäristössä käytössä ja Pre- ja PostAuthorize-annotaatioiden käyttö on salllittu.

Tämän selvityksen ohessa Combitech OY:lle kirjoitettua dokumentaatiota tullaan mahdollisesti hyödyntämään tulevaisuudessa metoditason valtuutusominaisuuksien implementoinnissa, jos tässä opinnäytetyössä kuvatun kaltaisissa ympäristöissä näitä valtuutusominaisuuksia halutaan hyödyntää.

6 Pohdinta

Omien SecuritySerivce-rajapintojen implementointi on todella helppoa ja niiden kutsuminen annotaatioiden sisällä on SpEL:n ansioista erittäin suoraviivaista. SpringSecurity tarjoaa myös kattavan tuen useille suosituille autentikaatio-ratkaisuille. Spring ja Spring Security ovat monipuolisia sovelluskehyksiä, jotka soveltuvat kaiken tasoisiin projekteihin. Java-sovellusten tietoturvan saralla Spring Securityllä ei ole juuri kilpailua. Muita vaihtoehtoja ovat Java EE Security ja Apache Shiro. Kaikki kolmet ovat hyvin samanlaisia tuotteita ja sisältävät pitkälti samat ominaisuudet. Spring Securityn ja Java EE Securityn dokumentaatiot ovat paljon Shiroa kattavampia. Spring Securityn isoin etu on sen suuri yhteisö. Lähes poikkeuksetta kaikkiin Spring Security -aiheisiin ongelmiin löytyy keskustelua eri palstoilta ja Stack Overflowsta. Jos projekti käyttää jo entuudestaan jotain muuta Spring tuoteperheen projektia on Spring Secuirity sovelluksen turvaamiseen melko selvä valinta.

Lähteet

About Thymeleaf. N.d. Tieto- ja FAQ-sivu Thymeleafista ja sen ominaisuuksista Thymeleafin kotisivuilla. Viitattu 11.7.2018. https://www.thymeleaf.org/faq.html.

A Brief History of NetBeans. N.d. Artikkeli NetBeans IDE:n historiasta sen kotisivuilla. Viitattu 11.7.2018. https://netbeans.org/about/history.html.

Alex, B., Taylor, L., Winch, R., Hillert, G., Grandja, J. & Bryant, J. 2017. Spring Security Reference. Viitattu 12.7.2018. https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/.

Knutson, M., Winch, R., & Mularien, P. 2017. Spring Security Third Edition. Birmingham: Packt Publishing.

Spring MVC view layer: Thymeleaf vs. JSP. N.d. Thymeleafin ja JSP:n eroja vertaileva artikkeli Thymeleafin kotisivuilla. Viitattu 11.7.2018. https://www.thymeleaf.org/doc/articles/thvsjsp.html.

Tietoja Meistä. N.d. Osio Combitech OY:n suomenkielisillä verkkosivuilla. Viitattu 17.8.2018. https://www.combitech.fi/tietoja.

Liitteet

Liite 1. Beer-entityluokka

```
package fi.combitech.beermanager.hibernate.db;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.MapsId;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.OneToOne;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;

/**
 *
 * @author udmama
 */
@Entity
@Table(name = "beer", catalog = "testDB", schema = "beermanager")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Beer.findAll", query = "SELECT b FROM Beer b")
    , @NamedQuery(name = "Beer.findById", query = "SELECT b FROM Beer b WHERE b.id = :id")
    , @NamedQuery(name = "Beer.findByName", query = "SELECT b FROM Beer b WHERE b.name = :name")
    , @NamedQuery(name = "Beer.findByBrewery", query = "SELECT b FROM Beer b WHERE b.brewery = :brewery")
    , @NamedQuery(name = "Beer.findByPrice", query = "SELECT b FROM Beer b WHERE b.price = :price")
    , @NamedQuery(name = "Beer.findByAbv", query = "SELECT b FROM Beer b WHERE b.abv = :abv")
    , @NamedQuery(name = "Beer.findByScore", query = "SELECT b FROM Beer b WHERE b.score = :score")
    , @NamedQuery(name = "Beer.findByDescription", query = "SELECT b FROM Beer b WHERE b.description = :description")})
public class Beer implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Size(min = 1, max = 50)
    @Column(name = "id")
    private String id;
    @Size(max = 50)
    @Column(name = "name")
    private String name;
    @Size(max = 50)
    @Column(name = "brewery")
    private String brewery;
    // @Max(value=?)  @Min(value=?)//if you know range of your decimal fields consider using these annotations to enforce field validation
    @Column(name = "price")
    private Double price;
    @Column(name = "abv")
    private Double abv;
    @Column(name = "score")
    private Integer score;
    @Size(max = 250)
    @Column(name = "description")
    private String description;
    @OneToOne
    @JoinTable(name="beer_user", schema = "beermanager",
    joinColumns = @JoinColumn(name="beer_id", referencedColumnName="id", unique=true, nullable=true, insertable=true, updatable=true),
            inverseJoinColumns = @JoinColumn(name="user_id", referencedColumnName="id"))
    private User user;

    public Beer() {
    }

    // Getters and Setters omitted...

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Beer)) {
            return false;
        }
        Beer other = (Beer) object;
        if ((this.id == null && other.id != null) || (this.id != null && !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "fi.combitech.beermanager.hibernate.db.Beer[ id=" + id + " ]";
    }

}
```

Liite 2. BeerDaoImpl 1/2

```
package fi.combitech.beermanager.dao;

import fi.combitech.beermanager.dto.BeerDto;
import fi.combitech.beermanager.dto.UserDto;
import fi.combitech.beermanager.hibernate.db.Beer;
import fi.combitech.beermanager.hibernate.db.User;
import fi.combitech.beermanager.services.UserContext;
import java.util.ArrayList;
import java.util.List;
import java.util.UUID;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Join;
import javax.persistence.criteria.Predicate;
import javax.persistence.criteria.Root;
import org.springframework.beans.factory.annotation.Autowired;

/**
 *
 * @author udmama
 */
public class BeerDaoImpl implements BeerDao {

    @PersistenceContext
    EntityManager entityManager;
    @Autowired
    private UserDao userDao;
    @Autowired
    private UserContext userContext;

    private static BeerDto dbToDomain(Beer dbBeer) {
        BeerDto beerDto = new BeerDto();
        beerDto.setId(dbBeer.getId());
        beerDto.setName(dbBeer.getName());
        beerDto.setBrewery(dbBeer.getBrewery());
        beerDto.setPrice(dbBeer.getPrice());
        beerDto.setAbv(dbBeer.getAbv());
        beerDto.setScore(dbBeer.getScore());
        beerDto.setDescription(dbBeer.getDescription());
        User user = dbBeer.getUser();
        UserDto userDto = UserDaoImpl.dbBeerToDomain(user);
        beerDto.setUser(userDto);

        return beerDto;
    }

    private static Beer domainToDb(BeerDto beerDto) {
        Beer beer = new Beer();
        beer.setId(beerDto.getId());
        beer.setName(beerDto.getName());
        beer.setBrewery(beerDto.getBrewery());
        beer.setPrice(beerDto.getPrice());
        beer.setAbv(beerDto.getAbv());
        beer.setScore(beerDto.getScore());
        beer.setDescription(beerDto.getDescription());
        UserDto user = beerDto.getUser();
        beer.setUser(UserDaoImpl.domainBeerToDb(beerDto.getUser()));

        return beer;
    }

    @Override
    public BeerDto saveBeer(BeerDto beer) {
        Beer dbBeer = domainToDb(beer);
        if (dbBeer.getId() == null) {
            dbBeer.setId(UUID.randomUUID().toString());
        }
        entityManager.merge(dbBeer);

        return beer;
    }

    @Override
    public List<BeerDto> findAll() {
        List<BeerDto> results = new ArrayList<>();

        try {
            TypedQuery<Beer> q = entityManager.createNamedQuery("Beer.findAll", Beer.class);
            List<Beer> beerList = q.getResultList();
            beerList.forEach((beer) -> {
                results.add(dbToDomain(beer));
            });
        } catch (Exception e) {

            e.getStackTrace();
        }
        return results;
    }
```

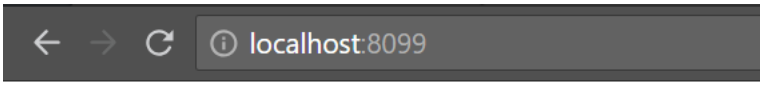
Liite 3. BeerDaoImpl 2/2

```
    @Override
    public BeerDto saveBeer(BeerDto beer) {
        Beer dbBeer = domainToDb(beer);
        if (dbBeer.getId() == null) {
            dbBeer.setId(UUID.randomUUID().toString());
        }
        entityManager.merge(dbBeer);

        return beer;
    }

    @Override
    public List<BeerDto> findAll() {
        List<BeerDto> results = new ArrayList<>();

        try {
            TypedQuery<Beer> q = entityManager.createNamedQuery("Beer.findAll", Beer.class);
            List<Beer> beerList = q.getResultList();
            beerList.forEach((beer) -> {
                results.add(dbToDomain(beer));
            });
        } catch (Exception e) {

            e.getStackTrace();
        }
        return results;
    }

    @Override
    public List<BeerDto> findByGroup() {

        List<BeerDto> results = new ArrayList<>();
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Beer> q = cb.createQuery(Beer.class);
        Root<Beer> r = q.from(Beer.class);



        Join<Beer, User> joinRoot = r.join("user");
        Predicate andPredicate = cb.like(cb.upper(joinRoot.get("userGroup").as(String.class)),
                (userContext.getCurrentUser().getUserGroup().toUpperCase()));
        q.select(r);
        q.where(cb.and(andPredicate));

        TypedQuery<Beer> query = entityManager.createQuery(q);
        List<Beer> res = query.getResultList();

        res.forEach((beer) -> {
            results.add(dbToDomain(beer));
        });

        return results;
    }

    @Override
    public List<BeerDto> findByUser() {

        List<BeerDto> results = new ArrayList<>();
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();
        CriteriaQuery<Beer> q = cb.createQuery(Beer.class);
        Root<Beer> r = q.from(Beer.class);



        Join<Beer, User> joinRoot = r.join("user");
        Predicate andPredicate = cb.like(cb.upper(joinRoot.get("username").as(String.class)),
                (userContext.getCurrentUser().getUsername().toUpperCase()));
        q.select(r);
        q.where(cb.and(andPredicate));

        TypedQuery<Beer> query = entityManager.createQuery(q);
        List<Beer> res = query.getResultList();

        res.forEach((beer) -> {
            results.add(dbToDomain(beer));
        });

        return results;
    }
}
```

Liite 4. BeerController

```
package fi.combitech.beermanager;

import fi.combitech.beermanager.forms.BeerForm;
import fi.combitech.beermanager.dto.BeerDto;
import fi.combitech.beermanager.services.BeerService;
import fi.combitech.beermanager.services.UserContext;
import java.util.List;
import javax.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

/**
 *
 * @author udmama
 */
@Controller
public class BeerController {

    @Autowired
    private BeerService beerService;
    @Autowired
    private UserContext userContext;

    @ModelAttribute("allBeers")
    public List<BeerDto> allBeers() {
        for (BeerDto beer : beerService.findByUsername()) {
        }
        return beerService.findByUsername();
    }

    @RequestMapping("/newbeer")
    public String newBeer(BeerForm beerForm) {
        return "newbeer";
    }

    @RequestMapping(value = "/newbeer", method = RequestMethod.POST)
    public String newBeer(@Valid BeerForm beerForm, BindingResult bindingResult) {
        BeerDto newBeer = new BeerDto();

        newBeer.setName(beerForm.getName());
        newBeer.setBrewery(beerForm.getBrewery());
        newBeer.setPrice(beerForm.getPrice());
        newBeer.setAbv(beerForm.getAbv());
        newBeer.setScore(beerForm.getScore());
        newBeer.setDescription(beerForm.getDescription());
        newBeer.setUser(userContext.getCurrentUser());

        beerService.saveBeer(newBeer);
        return "newbeer";
    }

    @RequestMapping("/beer_list")
    public String newBeer() {
        return "beer_list";
    }

    @RequestMapping(value="/beer_list", method=RequestMethod.POST, params="action=getByGroup")
    public String beersByGroup(Model model) {
        model.addAttribute("allBeers", beerService.findByGroup());
        return "beer_list";
    }

    @RequestMapping(value="/beer_list", method=RequestMethod.POST, params="action=getByUsername")
    public String beersByUsername(Model model) {
        model.addAttribute("allBeers", beerService.findByUsername());
        return "beer_list";
    }

    @RequestMapping(value="/beer_list", method=RequestMethod.POST, params="action=getAll")
    public String allBeers(Model model) {
        model.addAttribute("allBeers", beerService.findAll());
        return "beer_list";
    }

}
```

Liite 5. Aloitussivu



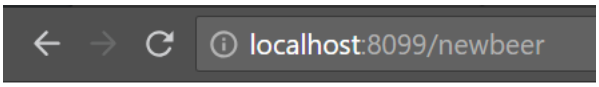
Home Login Register New beer Beer list
Hello,
admin
Logout

Welcome

hurdur: **Tue Aug 21 16:20:51 EEST 2018**

© 2018 Footer

Liite 6. Uuden oluen lisäyssivu



Home Login Register New beer Beer list
Hello,
admin
Logout

Add new beer

Name

Brewery

Price
0,0
Alcohol content
0,0
Score
0
Description

Add beer

Liite 7. Olutlista

Home Login Register New beer Beer list
Hello,
admin
Logout

Beer List

Get All | Get by group | Get my beers

Name	Brewery	Price	ABV	Score	Description
Punk IPA	Brew Dog	3.45	5.6	5	Hyvä aloittelijan IPA. Suosikki.
Karhu	Sinebrychoff	1.2	4.3	3	2. paras bulkkilager
Nobelaner	Laitilan Wirvoitusjuomatehdas	0.8	4.5	1	Halpa, mutta huono.
Sandels	Olvi	1.35	4.3	4	Paras bulkkilager